CS 458 A1-MILESTONE

Chia Ching Chuen |WATID: CCCHIA | STUDENT NUMBER: 20755359

Abstract

Explanation of exploit 1 and exploit 2

Sploit1.c

This exploit program will overflow the option -s of the program pwgen. As the program code for the parameter -s, accepts a user's input for the insertion of the string, it seems like a good place to start trying to exploit the program. Firstly, after looking through the code of the pwgen, in the code, I find that there exists a line of code :

```
args = parse_args(argc argv);
```

Inside of parse_args function, specifically, the line:

strcpy(args.salt, optarg);

as strcpy doesn't check whether the string variable is big enough for the string to be copied over, we are able to use this for a buffer overflow attack to change the return address after the function ends.

For now, the location of the return address, the amount of space to overwrite with NOPs and the shellcode before inserting the address of the location to jump to and the address to jump to is needed. first, I have to know where args.salt is located at specifically, using gdb, we are able to find the location at 0xffbfd7df by using the "print &args.salt" command.

As we checked, the distance from salt to the return address is roughly 557, we construct the buffer with a size of 557 bytes.

We first constructing the buffer with NOP, the shellcode and the address of args.salt repeatedly, to jump to in order. This will make the RA hopefully jump to the NOP or the shellcode.

By now, after compiling the program, and running it, a segmentation fault will be shown where my return address is a jumbled up of 0xffbfd7df address. After wrapping the address around and compiling it, the vulnerability is exploited, and a root shell is spawned.

To fix this vulnerability, simply use a check instruction before the strcpy instruction so that the string will stop copying whenever the buffer is full, which will prevent this buffer overflow exploit.

Sploit2.c

In print_usage function, we can see that when printing the help menu, its first uses the argument 0, which will be whatever command that was used to initialise the program and after which it prints the buffer with printf. However, in the printf statement, it directly prints the buffer without using any format string. With the format string %x, we can print variables on the stack, and as our argument is also on the stack, we just need to print enough variables, after which we will be able to rewrite the return address with the format string %n as it will write to the our address that was in our argument. As %n will insert the number of chars prints so far, we will have to print a very large number to overwrite the return address to the correct location. Another way, is to print our shellcode which in turn will execute the shellcode instruction and spawn a shell.

To exploit this, we will first need a payload which includes the shellcode and the NOPs, the address to the NOPs or shellcode and how far is the address of the buffer from the payload.

As the execve takes in another set of argument as the proper argument instead of using the filename as the argv[0], we are able to insert the payload into argv[0], and print the shellcode, doing so, it will allow us to achieve a rooted shell as the program always runs in root and thus after printing the shellcode, the return address will jump to one of the NOPs which will run the shellcode.

To fix this vulnerability, simply instead of printf(buffer), we will change the line to printf(“%s”,&buffer); which will print the buffer string the same.